1. Pr0100-Getting Started with Processing
2. Pr0110-Anatomy of the Processing Framework
3. Pr0120-Image Explorer
4. Pr0130-Introduction to Image Processing Algorithms
5. Pr0140-A space-wise linear pixel-modification algorithm

Pr0100-Getting Started with Processing
The purpose of this module is to help you get started programming in the Processing programming environment.

Table of Contents

Preface

This module is one in a collection of modules designed to teach you about the Processing open source programming environment.

Processing is a Java-based OOP programming environment designed for people who want to create images, animations, and interactions.

The purpose of this module is to help you get started programming in the Processing programming environment.

Not for beginners

This material is not for beginning programmers. The material in this collection assumes that you already have knowledge of programming fundamentals, preferably in the Java programming language. If that is not the case, I recommend that you first study the *Programming Fundamentals* material at Object-Oriented Programming (OOP) with Java and return to this collection once you understand that material.

The material in this collection concentrates on the *object-oriented* aspects of Java and the Processing environment.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the images while you are reading about them.

Images

- Image 1 . The Processing Development Environment (PDE).
- Image 2 . Output from the Cars sketch.
- Image 3 . JavaScript output mode.

Preview

In Processing, a computer program is called a *sketch* . Sketches are stored in the *Sketchbook* , which is a folder on your computer.

In this module, I will show you how to create an animated Processing sketch that can be run in any of three ways:

- Run directly from the PDE.
- Run in JavaScript mode.
- Export and run as an application.

Click here to see a live demonstration of the sketch in your HTML 5 compatible browser.

General background information

The true power of OOP

The true power of OOP lies not in the programming language, but in the class libraries that support that programming language. This is particularly true of Java OOP because, unlike C++, the Java programming language is lean, small, and compact.

Specialized class libraries

OO programmers working in industry often use a variety of specialized class libraries. The classes in those libraries tend to be specialized for the type of work being done; medical, database, shopping cart, graphics, games, etc.

Therefore, in addition to understanding critical OOP concepts, it is also important that aspiring OO programmers develop the ability to work with one or more class libraries in addition to the library that comes with the Java development kit. One of the main reasons that I am writing this

collection of modules is to give my OO students an opportunity to work with another class library.

Often the most difficult part to learning to use a new class library is learning how to read and interpret the documentation, and the Processing library is no exception.

The main documentation for Processing is accessible from the **Reference** and **Learning** links at the top of the Cover page . While extensive, this documentation is not presented in industry standard javadoc format. Documentation in standard javadoc format is available here .

Android and HTML 5 make Processing more relevant

The Processing environment has been around for a long time. It was initially developed to serve as a software sketchbook and to teach fundamentals of computer programming within a visual context. While it originally occupied a very narrow niche market, it has now become more relevant on a broad scale for two important reasons:

- Processing can be used to develop Android apps .
- Processing has a sister product named Processing.js that will automatically convert *(with some limitations)* Processing source code into JavaScript code suitable for use with the *canvas* element in HTML 5.

In other words, Processing directly supports two of the newest and hottest items in information technology; mobile apps and HTML 5 canvas.

Discussion and sample code

The Processing website provides a number of tutorials that explain important aspects of the Processing environment and also provide sample code that illustrates those aspects of the environment.

An integrated development environment (IDE)

As you will see when you study those tutorials, the Processing programming environment comes with its own integrated development environment *(known as the Processing Development Environment or PDE)*. A screen shot of the PDE is shown in Image 1 .

Image 1. The Processing Development Environment (PDE).

Figure

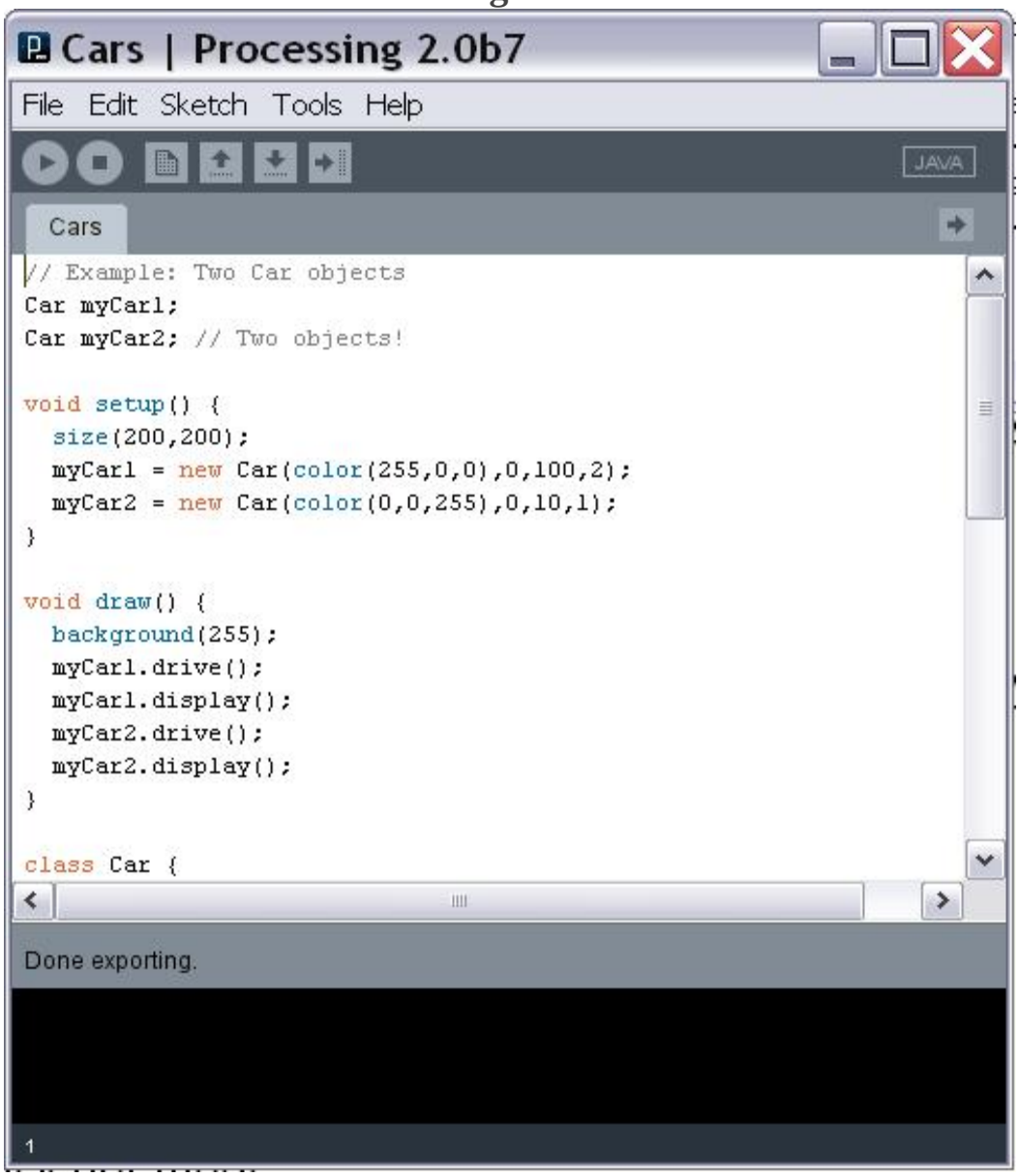


Image 1. The Processing Development Environment (PDE).

Click here to view an extensive discussion of the PDE.

While it is possible to develop Processing programs outside the PDE, that that presents several issues that I prefer not to deal with involving libraries, etc. Therefore, I will probably use the PDE in most of the modules in this collection.

The Getting Started tutorial

The Getting Started tutorial explains how to download and install the Processing environment on Windows, Mac, and Linux.

In addition, that tutorial shows you how to create your first sketch along with a lot of other useful information. Study it carefully.

Run the sketch

The Objects tutorial explains how to write a sketch in which you define your own class named **Car** and then instantiate and manipulate objects of that class.

I encourage you to copy the code from that tutorial into the PDE. *(Image 1 shows the beginning of that sketch.)*

Once you have copied the code into the PDE, you have three optional ways to run the sketch:

- Run directly from the PDE.
- Run in JavaScript mode.
- Export and run as an application.

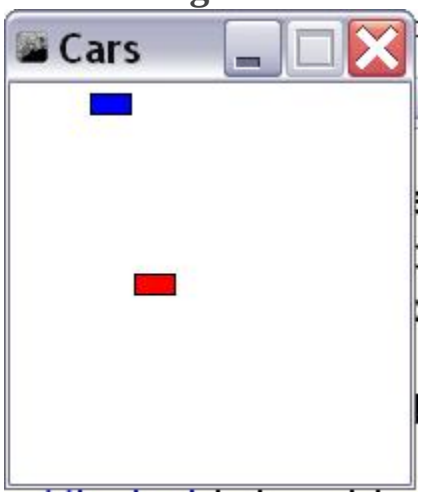
Run directly from the PDE

You can run the sketch directly from the PDE by either clicking the triangular arrow button shown near the upper-left corner in Image 1 , or by selecting **Run** from the **Sketch** menu.

When you do, a window should appear that looks something like Image 2 where the two animated rectangles move from left to right across the screen.

Output from the Cars sketch.

Figure



Output from the Cars sketch.

Run in JavaScript mode

Although it isn't easy to see, there is a button in the upper-right corner of the PDE in Image 1 that allows you to select an output mode. *(The default output mode is Java.)* Assuming that your default browser supports the *canvas* element in HTML 5, selecting the **JavaScript** output mode and clicking the **run** button will cause the sketch output to appear in your browser as shown in Image 3 . *(See the complication involving images.)*

Image 3. JavaScript output mode.

Figure

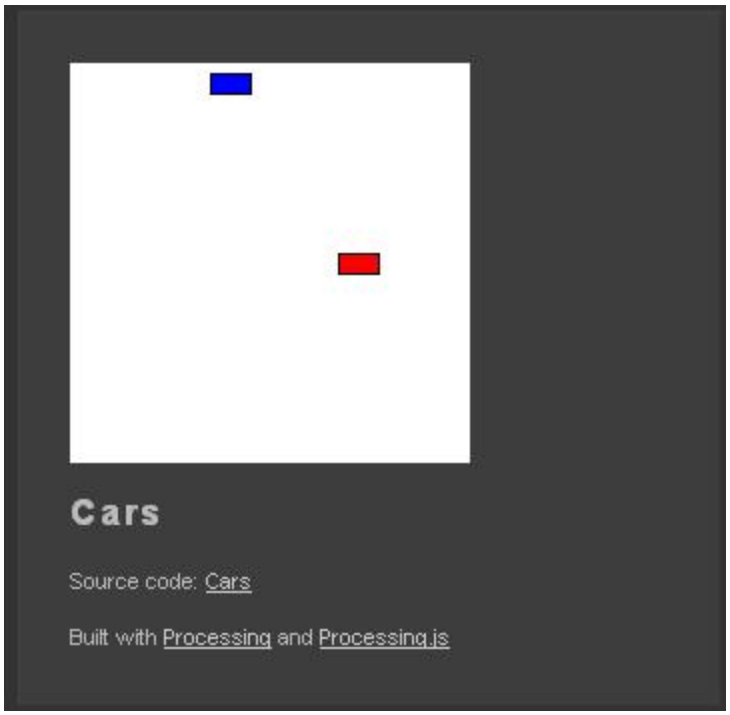


Image 3. JavaScript output mode.

A folder named web-export

That will also cause a folder named **web-export** to be created in your Processing sketchbook folder. The **web-export** folder will contain a file named **index.html** . Opening that file in your HTML 5 compatible browser will produce the output shown in Image 3 , which is the same output that you should see by clicking here .

A caution regarding JavaScript mode

Not all sketches that can be written in Processing can be successfully run in JavaScript mode. For example, there are two tabs on the references page:

- Standard Processing

- JavaScript (Processing.js)

If you examine the contents of those two tabs, you will see that some of the items that appear on the Standard Processing tab are "grayed out" on the JavaScript tab. If you plan to run your sketch in JavaScript mode, you should probably restrict yourself to using only the items on the JavaScript tab.

In addition, the Processing.js Quick Start - Processing Developer Edition page explains a number of restrictions that apply to the JavaScript mode.

Export and run as an application

Finally, selecting **Export Application** on the **File** menu, checking the **Windows** box, and clicking the **Export** button will cause two additional folders to be created in your sketchbook folder. *(Checking Mac OS X or Linux will cause different folders to be created.)*

- application.windows32
- application.windows64

Those folders contain everything you need to run your sketch as a stand-alone application on the selected platform.

Dealing with image and sound files

Although the sketch in this module doesn't involve image files, sketches in future modules will involve image files and possibly sound files as well.

The PDE expects image files and sound files to be located in a folder named **data** that is a child of the folder containing the files with the .pde extension. You should make certain that they are located there.

However, *(and this is unusual)*, when writing the Java code, you need to write it as though the image file is in the same folder as the pde file. In

other words, when referring to the file as a string, don't include the path to the **data** folder.

Summary

In this module, I showed you how to create an animated Processing sketch that can be run in any of three ways:

- Run directly from the PDE.
- Run in JavaScript mode.
- Export and run as an application.

Click here to see a live demonstration of the sketch in your HTML 5 compatible browser.

What's next?

In the next module, I will dissect and explain the structural anatomy of the Processing framework.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Pr0100-Getting Started with Processing
- File: Pr0100.htm
- Published: 02/25/13

Note: Disclaimers:

Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Pr0110-Anatomy of the Processing Framework
The purpose of this module is to explore and explain the anatomy of the Processing framework.

Table of Contents

Preface

This module is one in a collection of modules designed to teach you about the Processing open source programming environment.

Processing is a Java-based OOP programming environment for people who want to create images, animations, and interactions.

The purpose of this module is to explore and explain the anatomy of the Processing framework by dissecting and explaining an animated sketch.

Click here to view the JavaScript version of the sketch in your HTML 5 compatible browser.

Acknowledgement: The sketch that I will explain in this module is almost an exact copy of the sketch presented in Object Oriented Programming by Daniel Shiffman. I recommend that you take a look at what Shiffman has to say about this sketch as he develops it from a non-OO syntax to an OO syntax.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

Images

Listings

Preview

What you have learned

In the previous module, you learned how to download and install the Processing Development Environment *(PDE)* .

You also learned how to create an animated Processing sketch that can be run in any of three ways:

- Run directly from the PDE.
- Run in JavaScript mode.
- Export and run as an application.

What you will learn

In this module, you will learn about the structural anatomy of the Processing framework.

General background information

There's a good chance that your Java programming background is restricted to Java applications, *(which always have a **main** method)* or Java applets, *(which don't have a **main** method but do have something similar)* .

You may have noticed that the code in the previous module did not expose a **main** method or anything similar to a **main** method.

A software framework

The *Processing Development Environment* is really a software framework. What do I mean by that? Here is part of what Wikipedia has to say about a software framework:

A software framework, in computer programming, is an abstraction in which common code providing generic functionality can be selectively overridden or specialized by user code providing specific functionality.

Frameworks are a special case of software libraries in that they are reusable abstractions of code wrapped in a well-defined API, yet they contain some key distinguishing features that separate them from normal libraries.

Software frameworks have these distinguishing features that separate them from libraries or normal user applications:

1. ***inversion of control*** *- In a framework, unlike in libraries or normal user applications, the overall program's flow of control is not dictated by the caller, but by the framework.*
2. ***default behavior*** *- A framework has a default behavior. This default behavior must actually be some useful behavior and not a series of no-ops.*
3. ***extensibility*** *- A framework can be extended by the user by selective overriding of framework code in order to provide specific functionality*
4. ***non-modifiable framework code*** *- The framework code, in general should not normally be modified by the user. Users can extend the framework, but normally should not modify its code.*

In short, a software framework is a computer program that helps you to write computer programs.

The Processing PDE fits all of these characteristics of a framework.

Inversion of control

By default, the overall flow of control of a Processing sketch is beyond the control of the programmer. When the sketch starts running, the **setup** method will be called once and *(by default)* the **draw** method will be called repeatedly at a default rate of 60 calls per second.

A small window will be displayed in the default Java display mode. If you haven't overridden the **draw** method to control the contents of that window, it will look something like that shown in **Image 1** .
Image 1. Default display window.

Figure

Image 1. Default display window.

Create the body of a class

Image 2 shows a screen shot of the PDE with program code showing in the leftmost tab.
Image 2. The PDE.

Figure

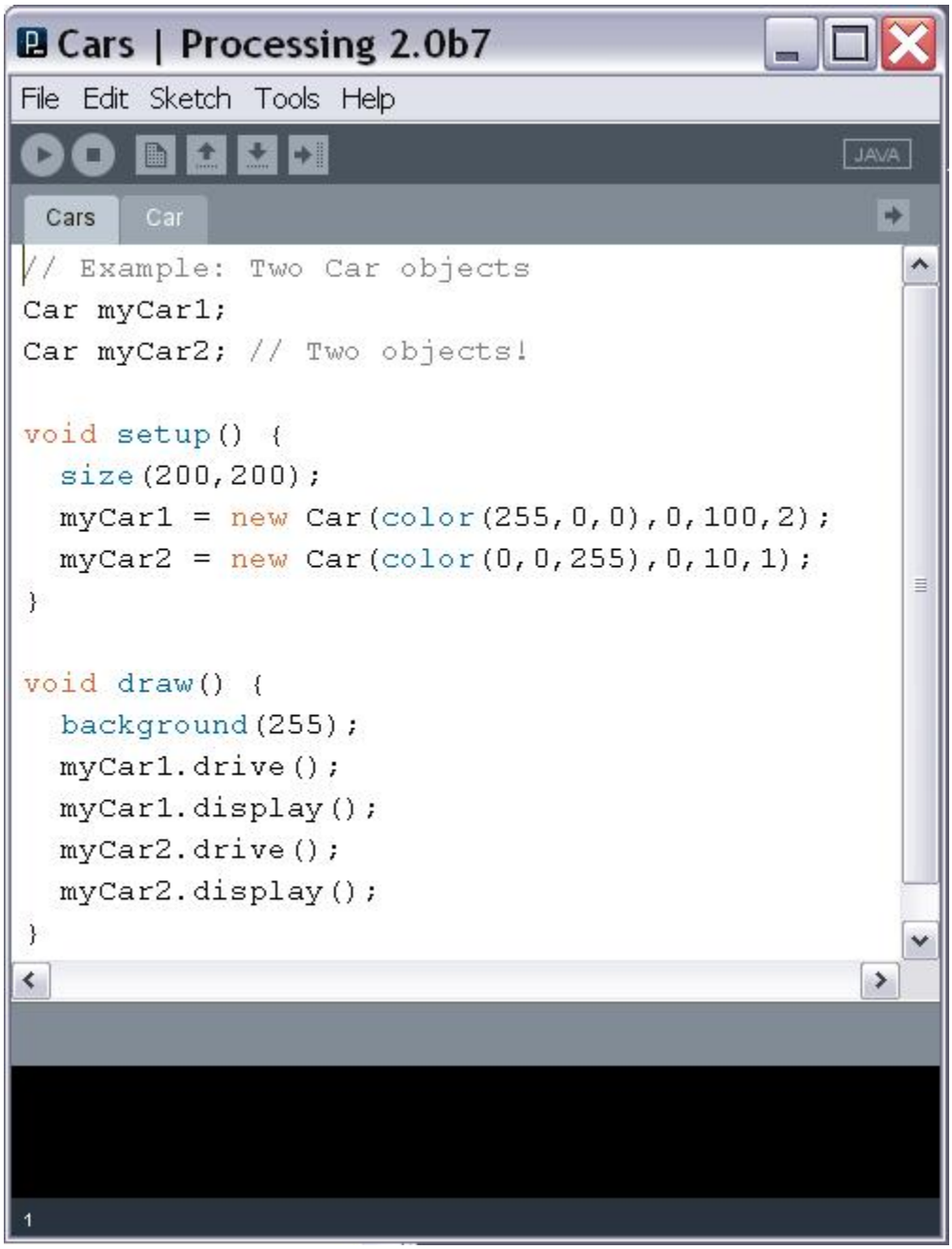


Image 2. The PDE.

When you create a new sketch in the PDE and enter code in the leftmost tab, you are actually writing the body of a new class with a class name that

matches the name of the tab. The code that you see in Image 2 , for example, constitutes the body of a new class named **Cars** .

Override setup and draw methods

In general, you need to override the methods named **setup** and **draw** in the body of this class. It is the code that you write in these two overridden methods that controls the behavior of the sketch.

The **setup** method will be called once when you click the **run** button. You should place any code that needs to be executed to initialize the state of the sketch in this method. Often this code will initialize variables, instantiate new objects, etc.

After that, by default, the **draw** method will be called repeatedly until you terminate execution of the sketch. You need to write code in the **draw** method that controls the ongoing behavior of the sketch. Often, this code will call methods on existing objects, modify the contents of variables, instantiate and call methods on new objects, etc.

Default behavior

As I mentioned above, the default behavior of the sketch is to call the **setup** method once and to repeatedly call the **draw** method until the sketch is terminated, regardless of whether or not the **setup** and **draw** methods are overridden. If these methods are not overridden, the sketch will produce an output similar to that shown in Image 1 .

Extensibility

As you will see later, for the sketch shown in Image 2 , the framework has been extended by overriding the **setup** and **draw** methods, and by defining a new class named **Car** . The code for the class named **Car** is on the hidden tab named **Car** in Image 2 and is also shown in Listing 1 .

Listing 1. Code for the class named Car.

Figure

```
class Car {
  color c;
  float xpos;
  float ypos;
  float xspeed;

  // The Constructor is defined with arguments.
  Car(color tempC,
      float tempXpos,
      float tempYpos,
      float tempXspeed) {
    c = tempC;
    xpos = tempXpos;
    ypos = tempYpos;
    xspeed = tempXspeed;
  }

  void display() {
    stroke(0);
    fill(c);
    rectMode(CENTER);
    rect(xpos,ypos,20,10);
  }

  void drive() {
    xpos = xpos + xspeed;
    if (xpos > width) {
      xpos = 0;
    }
  }
}
```

Listing 1. Code for the class named Car.

Non-modifiable framework code

While we can extend the Processing framework by overriding methods, defining new methods, defining new classes, etc., we cannot, by default, modify the framework code.

Overall behavior of the Processing framework

Without seeing the source code, it appears that when we click the **Run** button in Java mode, the Processing framework:

- Instantiates a new object of a class that we define in the leftmost tab of the PDE.
- Calls the method named **setup** , *(which may or may not be overridden in the new class)* belonging to that object once and only once.
- Calls the method named **draw** , *(which may or may not be overridden in the new class)* belonging to that object repeatedly at a default rate of 60 calls per second until the sketch is terminated.

We can

- control how often the **draw** method is executed by calling the **frameRate** method,
- prevent the **draw** method from being executed repeatedly by calling the **noLoop** method,
- cause the **draw** method to be executed repeatedly by calling the **loop** method
- cause the **draw** method to be executed once by calling the **redraw** method.

Programs or sketches that we write to run under the Processing framework must be compatible with the behavior described above.

Discussion and sample code

Now it's time to put the theory aside and take a look at an actual sketch.

Listing 1 shows the definition of a class named **Car** . This class is defined on the hidden tab in Image 2 .

Listing 2 shows the definition of the class named **Cars** in the leftmost tab of Image 2 . I am repeating it here so that you can copy and paste it into your PDE.

Listing 2. Code for the class named Cars.

Figure

```
Car myCar1;//instance variables
Car myCar2;

void setup() {
  size(200,200);
  myCar1 = new Car(color(255,0,0),0,100,2);
  myCar2 = new Car(color(0,0,255),0,10,1);
}//end setup method

void draw() {
  background(255);
  myCar1.drive();
  myCar1.display();
  myCar2.drive();
  myCar2.display();
}//end draw method
```

Listing 2. Code for the class named Cars.

A subclass of PApplet

You might think of the class that is defined in Listing 2 as the driver class. You will see later that it is a subclass of the Processing class named **PApplet** .

The name of the new class

When we create a new sketch, this class has a default name something like **sketch_130223a** .. We can change the name of the class when we select **Save** or **Save As...** from the **File** menu of the PDE.

Saving the sketch causes a sketchbook folder to be created containing a file with a name like **sketch_130223a.pde** , or whatever name we specify. The name of the sketchbook folder is the same as the name of the class.

For the case of Image 2 , the sketchbook folder is named **Cars** and the file is named **Cars.pde** . *(The file named Cars.pde is simply a text file containing Java source code. If you were working outside the PDE, you could name it Cars.java.)*

Inherited methods

We can see from Listing 2 that our new class inherits a method named **setup** , a method named **draw** , and a method named **size** . The code in Listing 2 overrides the **setup** and **draw** methods and calls the **size** method without overriding it.

The class named **PApplet** defines a huge number of methods. Those methods can be called from within our class named **Cars** *(by novice programmers)* without knowledge of anything related to object-oriented

programming. They look pretty-much like global function calls from the dark ages.

A function library

In effect, the authors of Processing have created a huge function library, *(reminiscent of the function libraries of the dark ages of Pascal and C)* . Novice programmers can use that library to create sketches using a procedural programming style *(as opposed to an object-oriented programming style)* ..

A tricycle and a bicycle

While I don't think this is a good way to teach people how to program, I am impressed that the authors of Processing were able to pull it off.

In my opinion, learning to program this way is like trying to learn how to ride a bicycle by riding a tricycle. *(I'm not talking about a bicycle with removable training wheels. I'm talking about a vehicle with three fixed wheels.)*

As long as that tricycle *(the PDE)* will get you where you need to go, everything is okay. However, without special effort on your part, you will be ill-equipped to put the PDE aside and ride the bicycle *(write object-oriented code*) if this is how you learn to program.

Understanding the anatomy of the framework

That having been said, by understanding the anatomy of the framework, it is possible for novices and experienced programmers alike

- to think in object-oriented terms,
- to write serious object-oriented programs using the PDE, and
- to take advantage of the many libraries supported by Processing.

The ability to effectively use multiple libraries is one of the hallmarks of a successful OO programmer.

The draw method

The description of the **draw** method *(that is inherited from the **PApplet** class)* reads partially as follows:

"Called directly after setup() and continuously executes the lines of code contained inside its block until the program is stopped or noLoop() is called.

The draw() function is called automatically and should never be called explicitly.

It should always be controlled with noLoop(), redraw() and loop().

After noLoop() stops the code in draw() from executing, redraw() causes the code inside draw() to execute once and loop() will cause the code inside draw() to execute continuously again.

The number of times draw() executes in each second may be controlled with frameRate() function.

There can only be one draw() function for each sketch and draw() must exist if you want the code to run continuously or to process events such as mousePressed(). Sometimes, you might have an empty call to draw() in your program ..."

Default behavior of the framework

We know that the default behavior of the framework is to display a window on the screen similar to Image 1 if we don't override the **draw** method.

The same thing happens if we do override the **draw** method but leave the body empty. Therefore, the physical creation of the display window is being accomplished in the framework by code outside the **draw** method. The **draw** method simply controls the contents of the display window.

Code in the Cars class

Instance variables of type Car

The code in the **Cars** class *(see Listing 2)* begins by declaring two instance variables of the class named **Car** . By default, these two instance variables are initialized to null, meaning that they don't contain references to objects.

The inherited size method

The **setup** method begins by calling the method named **size** , which is inherited from the **PApplet** class. The documentation for the size method reads partially as follows:

*"Defines the dimension of the display window in units of pixels. The **size()** function must be the first line in **setup()** . If **size()** is not used, the default size of the window is 100x100 pixels. The system variables **width** and **height** are set by the parameters passed to this function.*

*Do not use variables as the parameters to **size()** function, because it will cause problems when exporting your sketch. When variables are used, the dimensions of your sketch cannot be determined during export. Instead, employ numeric values in the **size()** statement, and then use the built-in **width** and **height** variables inside your program when the dimensions of the display window are needed.*

*The **size()** function can only be used once inside a sketch, and cannot be used for resizing."*

The display window

In case you were wondering about the type of object shown in Image 1 , here is some information from the PApplet documentation.

"This class extends Applet instead of JApplet because 1) historically, we supported Java 1.1, which does not include Swing (without an additional, sizable, download), and 2) Swing is a bloated piece of crap. A Processing applet is a heavyweight AWT component, and can be used the same as any other AWT component, with or without Swing.

Similarly, Processing runs in a Frame and not a JFrame. However, there's nothing to prevent you from embedding a PApplet into a JFrame, it's just that the base version uses a regular AWT frame because there's simply no need for Swing in that context. If people want to use Swing, they can embed themselves as they wish."

Stated differently, the *display window* that you see in Image 1 is a standard **java.awt.Frame** object.

Instantiate Car objects

After setting the size of the display window, the **setup** method in Listing 2 instantiates two objects of the **Car** class and saves the object's references in the instance variables named **myCar1** and **myCar2** .

We haven't discussed the constructor for the **Car** class yet, but we can see in Listing 2 that the constructor for one **Car** object is passed a *color* value of pure red and the constructor for the other **Car** object is passed a *color* value of pure blue.

Let's talk about color

If you are an experienced Java programmer, what I am about to tell you is going to seem really strange.

Primitive types

As an experienced Java programmer, you know that the Java language defines the following eight primitive types:

- boolean
- char
- double
- float
- byte
- short
- int
- long

However, the Processing reference lists the following eight primitive types:

- boolean
- byte
- char
- color
- double
- float
- int
- long

You will note that the primitive **short** type does not appear in this list. You will also note that a new primitive type named **color** has been added to the list.

It is easy enough to demonstrate that the **short** type can be used in Processing sketches. Perhaps the omission of **short** from the list of primitive types was simply an oversight.

I won't even speculate on how the authors of Processing created something named **color** that behaves like a primitive type, but they obviously did.

The primitive color type

As an experienced Java programmer, you know that the Java programming language does not have a primitive type named **color** . However, the standard Java class library does have a class named **Color** *(distinguished by an upper-case C)* .

Here is part of what the Processing documentation has to say about the new primitive color type:

"Datatype for storing color values. Colors may be assigned with get() and color() or they may be specified directly using hexadecimal notation such as #FFCC00 or 0xFFFFCCOO.

Using print() or println() on a color will produce strange results (usually negative numbers) because of the way colors are stored in memory. A better technique is to use the hex() function to format the color data, or use the red(), green(), and blue() functions to get individual values and print those.

The hue(), saturation(), and brightness() functions work in a similar fashion. To extract red, green, and blue values more quickly (for instance when analyzing an image or a frame of video), use bit shifting.

Values can also be created using web color notation. For example, "color c = #006699".

Web color notation only works for opaque colors. To define a color with an alpha value, you can either use the color() function, or use hexadecimal notation. For hex notation, prefix the values with "0x", for instance "color c = 0xCC006699". In that example, CC (the hex value of 204) is the alpha value, and the remainder is identical to a web color. Note the alpha value is first in the hexadecimal notation (but last when used with the color() function, or functions like fill() and stroke()).

From a technical standpoint, colors are 32 bits of information ordered as AAAAAAAARRRRRRRRGGGGGGGGBBBBBBBB where the A's contain the alpha value, the R's are the red value, G's are green, and B's are blue. Each component is 8 bits (a number between 0 and 255). These values can be manipulated with bit shifting."

The color method

The word **color** is also used as the name of a method that is inherited from the **PApplet** class. Here is part of what the documentation has to say about the color method.

"Creates colors for storing in variables of the color datatype. The parameters are interpreted as RGB or HSB values depending on the current colorMode(). The default mode is RGB values from 0 to 255 and, therefore, the function call color(255, 204, 0) will return a bright yellow color...

Note that if only one value is provided to color(), it will be interpreted as a grayscale value. Add a second value, and it will be used for alpha transparency. When three values are specified, they are interpreted as either RGB or HSB values. Adding a fourth value applies alpha transparency."

Therefore, the calls to the **color** method in the two calls to the **Car** constructor in Listing 2 create and pass **color** values for pure red and pure blue as parameters to the constructor.

Termination of the setup method

That completes the discussion of the **setup** method in Listing 2 . When the method terminates, the size of the display window has been set to 200 x 200 pixels. Two objects of the **Car** class have been instantiated and those object's references have been saved in the instance variables named **myCar1** and **myCar2** .

Without having examined the constructor code for the **Car** class yet, about all we know about the **Car** objects is that a red **color** value was passed to the constructor for one object and a blue **color** value was passed to the constructor for the other object.

This looks like a good time to examine the constructor for the **Car** class in order to understand the meaning of the other constructor parameters in Listing 2 .

Constructor for the Car class

The code fragment in Listing 3 shows the beginning of the **Car** class down through the constructor. *(The code fragment in Listing 3 was extracted from Listing 1 to make it easier to discuss.)*

Listing 3. Beginning of the Car class.

Figure

```
class Car {
  color c;
  float xpos;
  float ypos;
  float xspeed;

  // The Constructor is defined with arguments.
  Car(color tempC,
      float tempXpos,
      float tempYpos,
      float tempXspeed) {
    c = tempC;
    xpos = tempXpos;
    ypos = tempYpos;
    xspeed = tempXspeed;
  }
```

Listing 3. Beginning of the Car class.

Instance variables

The **car** class begins by declaring four instance variables to hold values for:

- A color
- An x-coordinate value

- A y-coordinate value
- A speed value

The constructor parameters

Comparing the order of the constructor parameters in Listing 3 with the parameter values in Listing 2 tells us that the constructor for the first instantiated object of the **Car** class received the **following values** :

- A color = red
- An x-coordinate value = 0
- A y-coordinate value = 100
- A speed value = 2

Similarly, the constructor for the second instantiated object of the **Car** class received the following values:

- A color = blue
- An x-coordinate value = 0
- A y-coordinate value = 10
- A speed value = 1

The constructor body

The four statements in the body of the constructor in Listing 3 simply save the values of the incoming parameters in the corresponding instance variables discussed above .

Interpretation of the constructor parameters

Image 3 shows a screen shot of the display window while the sketch is running.

Image 3. Screen shot of the sketch during execution.

Figure

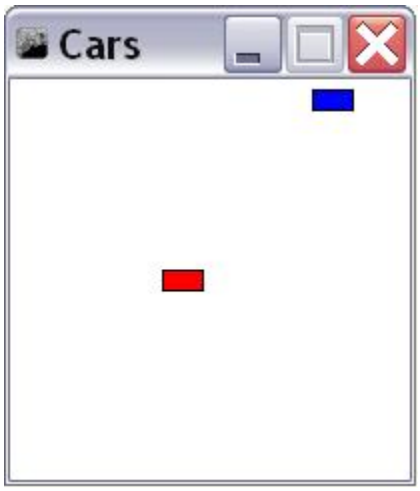


Image 3. Screen shot of the sketch during execution.

Looking at Image 3 , you can probably guess that the **color** values passed to the constructor are used to establish the visible color of each of the two rectangles.

Recall that this sketch is an animation where the rectangles move across the screen from left to right at different speeds You will see later that the coordinate values received by the constructor specify the initial positions of the rectangles and that the y-coordinate values never change.

Given that the 0,0 origin is at the upper left, you should be able to correlate the vertical position *(y-coordinate value)* of each rectangle with the color of the rectangle.

Because Image 3 is a screen shot, there is nothing in Image 3 that corresponds to the speed parameter in Listing 3 . You will see later that this parameter is used to establish the speed with which each rectangle moves across the display window. Stated differently, each rectangle appears to move across the display window in incremental steps. The speed value is used by the **drive** method to compute the size of each incremental step.

The draw method of the Cars class

Returning now to the **Cars** class, the code fragment in Listing 4 shows the **draw** method in its entirety.

Listing 4. The draw method of the Cars class.

Figure

```
void draw() {
  background(255);

  myCar1.drive();
  myCar1.display();

  myCar2.drive();
  myCar2.display();
}
```

Listing 4. The draw method of the Cars class.

Recall that by default, the **draw** method is called repeatedly 60 times per second. Nothing was done in this sketch to change that.

Set the background color

The **draw** method begins by calling the inherited **background** method to erase everything on the display window and set its color to white.

There are seven overloaded versions of the **background** method where different parameter types and order are used to produce different behavior.

Here is part of what the documentation has to say about the background method.

"The background() function sets the color used for the background of the Processing window. The default background is light gray. background() is typically used within the draw() function to clear the display window at the beginning of each frame.

An image can also be used as the background for a sketch, although the image's width and height must match that of the sketch window. To resize an image to the size of the sketch window, use image.resize(width, height).

Images used with background() will ignore the current tint() setting.

It is not possible to use transparency (alpha) with background colors on the main drawing surface; you can achieve the same effect with createGraphics()."

The version of the background method that is called in Listing 4 takes a single parameter. That versions sets the background color to a gray value between white and black. A parameter value of 255 produces white and a parameter value of 0 produces black.

Call methods on the Car objects

If you are new to OOP, you may be unfamiliar with the following syntax:

myCar1.drive();

Basically this means:

- Go find the object whose reference is stored in the reference variable named **myCar1** .
- Knock on the object's door and ask it to execute its method named **drive** .

Each time the **draw** method is executed, it asks each **Car** object to first execute its **drive** method and then execute its **display** method.

At this point, we don't know what that means exactly because we haven't examined the behavior of the methods named **drive** and **display** belonging to the **Car** objects.

The display method of the Car class

Switching again to the **Car** class, the code fragment in Listing 5 shows the display method of the **Car** class.

Listing 5. The display method of the Car class

Figure

```
void display() {
  stroke(0);
  fill(c);
  rectMode(CENTER);
  rect(xpos,ypos,20,10);
}//end method named display
```

Listing 5. The display method of the Car class

Note that all four methods that are called in the body of the **display** method are inherited from the class named **PApplet** .

The stroke method

Six overloaded versions of the **stroke** method are inherited from the **PApplet** class into the **Cars** class.

*(Methods inherited into the **Cars** class are accessible by objects of the **Car** class because the **Car** class is converted to an inner class of the **Cars** class when the program is compiled by the Processing environment. However,*

inner classes is a relatively advanced topic that is beyond the scope of this module.)

Here is some of what the documentation has to say about the stroke method.

"Sets the color used to draw lines and borders around shapes. This color is either specified in terms of the RGB or HSB color depending on the current colorMode() (the default color space is RGB, with each value in the range from 0 to 255).

When using hexadecimal notation to specify a color, use "#" or "0x" before the values (e.g. #CCFFAA, 0xFFCCFFAA). The # syntax uses six digits to specify a color (the way colors are specified in HTML and CSS).

When using the hexadecimal notation starting with "0x", the hexadecimal value must be specified with eight characters; the first two characters define the alpha component and the remainder the red, green, and blue components.

The value for the gray parameter must be less than or equal to the current maximum value as specified by colorMode(). The default maximum value is 255."

The version of the display method called in Listing 5 requires a single parameter of type **int** . This version apparently treats 24 bits of the 32-bit int value as the red, green, and blue color values and ignores the remaining 8 bits.

The parameter value of 0 passed to the stroke method in Listing 5 specifies a black stroke. This causes the rectangles shown in Image 3 to have black borders.

The fill method

The next method called inside the **display** method in Listing 5 is the method named **fill** , which is also inherited from the **PApplet** class. The

value stored in the **color** instance variable named **c** is passed as a parameter to the **fill** method.

The **Cars** class inherits six overloaded versions of the **fill** method from the **PApplet** class. Here is part of what the documentation has to say about the fill method.

"Sets the color used to fill shapes. For example, if you run fill(204, 102, 0), all subsequent shapes will be filled with orange. This color is either specified in terms of the RGB or HSB color depending on the current colorMode(). (The default color space is RGB, with each value in the range from 0 to 255.)

When using hexadecimal notation to specify a color, use "#" or "0x" before the values (e.g., #CCFFAA or 0xFFCCFFAA). The # syntax uses six digits to specify a color (just as colors are typically specified in HTML and CSS). When using the hexadecimal notation starting with "0x", the hexadecimal value must be specified with eight characters; the first two characters define the alpha component, and the remainder define the red, green, and blue components.

The value for the "gray" parameter must be less than or equal to the current maximum value as specified by colorMode(). The default maximum value is 255.

To change the color of an image or a texture, use tint()."

The version of the fill method called in Listing 5 requires a single parameter of type **color** or a hex value. In this case, a value of type **color** is passed as a parameter.

The call to the **fill** method in Listing 5 causes one of the rectangles in Image 3 to be filled with a red color and causes the other rectangle to be filled with a blue color. Note however that the rectangles don't exist when the **fill** method returns. The call to the **fill** method simply determines what the fill color will be when shapes such as rectangles are created later.

The rectMode method

To make a long story short, the call to the **rectMode** method in Listing 5 , passing CENTER as a parameter specifies that the position parameters used later to specify the location of a rectangle will be interpreted as the center of the rectangle, as opposed to the upper-left corner of the rectangle.

I will leave it as an exercise for the student to investigate this further in the documentation for the rectMode method.

The rect method

We have finally arrived at the ultimate purpose of the **display** method, which is

- to draw a rectangle
- with a specific width and height,
- at a specific location,
- with specific border and fill colors,
- on the display window.

The **Cars** class inherits three overloaded versions of the **rect** method from the **PApplet** class. Here is part of what the documentation has to say about the rect method.

Draws a rectangle to the screen. A rectangle is a four-sided shape with every angle at ninety degrees. By default, the first two parameters set the location of the upper-left corner, the third sets the width, and the fourth sets the height. The way these parameters are interpreted, however, may be changed with the rectMode() function.

To draw a rounded rectangle, add a fifth parameter, which is used as the radius value for all four corners.

To use a different radius value for each corner, include eight parameters. When using eight parameters, the latter four set the radius of the arc at

each corner separately, starting with the top-left corner and moving clockwise around the rectangle.

The version of the **rect** method called in Listing 5 requires four parameters.

The first two parameters passed to the **rect** method in Listing 5 are the x and y coordinate values for the center of the rectangle. These values are stored in the instance variables shown in Listing 3 . You will see later that the x-coordinate values are modified by the **drive** method so that the rectangle will be drawn at a different position each time the **display** method is called.

The last two parameters passed to the **rect** method specify that the rectangle will have a width of 20 pixels and a height of 10 pixels.

That completes the discussion of the **display** method shown in Listing 5 .

The drive method of the Car class

The drive method of the **Car** class is shown in its entirety in Listing 6 .
Listing 6. The drive method of the Car class.
Figure

```
  void drive() {
    xpos = xpos + xspeed;
    if (xpos > width) {
      xpos = 0;
    }//end if statement
  }//end method named drive
}//end class named Car
```

Listing 6. The drive method of the Car class.

The purpose of the **drive** method is to increase the x-coordinate position value stored in the instance variable in Listing 3 each time the method is called. However, when the value of the x-coordinate *(the center of the rectangle)* goes off the right side of the display window in Image 3 , the x-coordinate value is reset to 0.

Putting it all together

Thus, the process of (1) erasing the display window, (2) calling the **Car** object's **display** method, and (3) calling the **Car** object's **drive** method each time the **draw** method is called in Listing 4 , causes the rectangles to appear to move from left to right across the display window and then to start over on the left side when they reach the right side.

There you have it; a detailed object-oriented explanation of an animated Processing sketch written using OO techniques.

Click here to view the JavaScript version of this sketch in your HTML 5 compatible browser.

Run the program

I encourage you to copy the code from Listing 1 and Listing 2 into your PDE, being careful to copy the code from Listing 2 into the leftmost PDE tab.

Run the sketch and observe the results. Experiment with the code. Make changes, run the sketch again, and observe the results of your changes. Change the speed. Change the color. Change the initial position. Change the size and background color of the display window. Make certain that you can explain why your changes behave as they do.

Don't forget to also create and run the JavaScript version of your sketch in your HTML 5 compatible browser.

If you have a programmable Android device, try creating and running the Android version of your sketch in your Android device.

Summary

In this module, I taught you about the anatomy of the Processing framework by dissecting and explaining an animated sketch.

Click here to view the JavaScript version of this sketch in your HTML 5 compatible browser.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Pr0110-Anatomy of the Processing Framework
- File: Pr0110.htm
- Published: 02/22/13

nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

-end-

Pr0120-Image Explorer
The purpose of this module is teach you how to write an image explorer sketch that can be used determine the coordinates and RGB color values of any pixel in an image by pointing to the pixel with a mouse. The sketch can also be used to determine the dimensions of the image.

Table of Contents

Preface

This module is one in a collection of modules designed to teach you about the Processing open source programming environment.

Processing is a Java-based OOP programming environment for people who want to create images, animations, and interactions.

The purpose of this module is to teach you how to write an *image explorer* sketch that can be used determine the coordinates and RGB color values of any pixel in an image by pointing to the pixel with a mouse. The sketch can also be used to determine the dimensions of the image.

Click here to view the JavaScript version of the sketch discussed in this module in your HTML 5 compatible browser.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

Images

- Image 1 . Normal output.
- Image 2 . Output for image too wide.
- Image 3 . Output for image too tall.
- Image 4 . Output produced by a common mathematical algorithm.

Listings

Preview

In this module, I will teach you how to write a Processing sketch that can be used to determine the following information by pointing to a location in an image and pressing a mouse button:

- Width and height of an image.
- X and Y coordinates of the mouse pointer.
- RGB color values for the pixel at the mouse pointer.

Normal output from the sketch

Normal output from this sketch is shown in the screen capture image in Image 1 . *(Note that a screen capture image does not show the mouse pointer.)*

Image 1. Normal output.

Figure

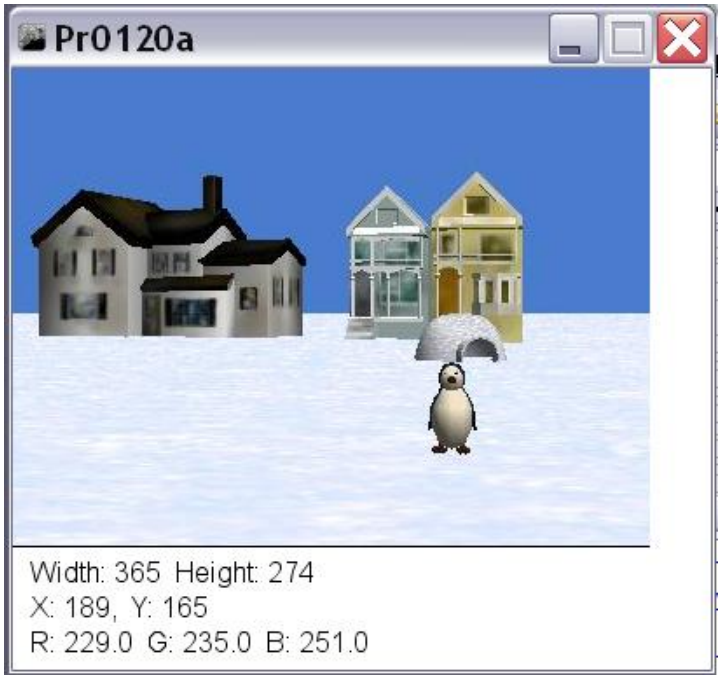


Image 1. Normal output.

Output when the image is too wide

Image 2 shows the output produced by the program when the width of the image is wider than the width of the display window.

Image 2. Output for image too wide.

Figure

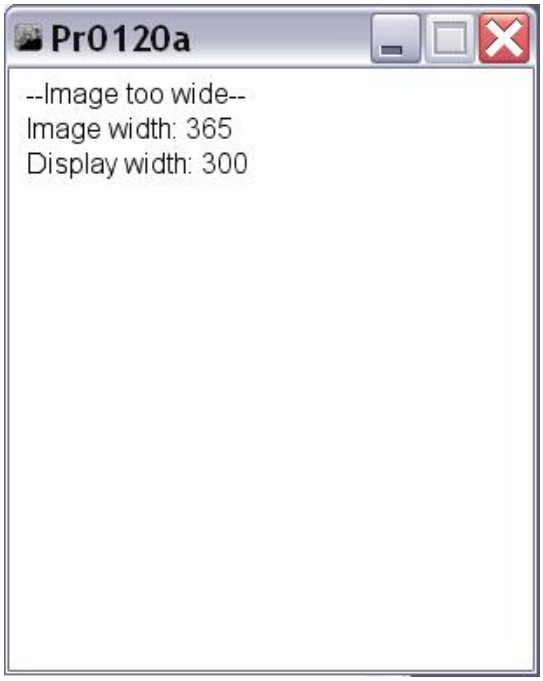


Image 2. Output for image too wide.

Output when the image is too tall

Image 3 shows the output that is produced when the height of the image is greater than the height of the display window.

Image 3. Output for image too tall.

Figure

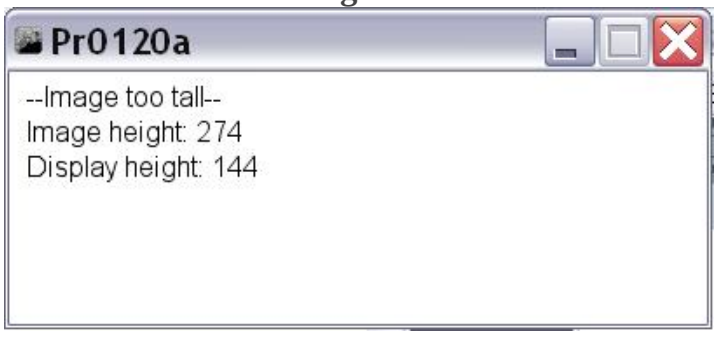


Image 3. Output for image too tall.

General background information

Students in my courses are frequently required to

1. Examine the input and output images from an image processing algorithm.
2. Deduce the nature of the algorithm required to transform the input image into the output image.
3. Implement the algorithm in program code.
4. Demonstrate the correctness of their solution.

Items 2 and 4 often require the ability to make precise color measurements on a pixel-by-pixel basis. The sketch that I will explain in this module provides the ability to make such measurements.

For example, the image shown in Image 4 was created by applying a common mathematical algorithm to the image shown in Image 1 .

Image 4. Output produced by a common mathematical algorithm.

Figure

Image 4. Output produced by a common mathematical algorithm.

The RGB color values for both images were displayed at the same pixel location. A student would be expected to compare the color values at several locations, deduce, and

then implement the algorithm in program code.

Discussion and sample code

This sketch consists of two Java classes. The code for the *driver* class named **Pr0120a** *(the one in the leftmost panel of the PDE)* is provided in Listing 10 . The code for the *runner* class named **Pr0120aRunner** is provided in Listing 11 .

I will break this code down and explain it in fragments.

As shown in Image 1 , this sketch displays the coordinates of the mouse pointer along with the RGB color values of the pixel at the mouse pointer whenever the user points to a location inside the display window and presses a mouse button.

It is easy to modify the sketch to cause it to display that information without the requirement to press a mouse button.

The sketch also displays the width and the height of the image.

The sketch displays an error message in place of the image if the image is wider or taller than the output display window.

The class named Pr0120a

Beginning of the class

Listing 1 shows the special text that you must include in the code if you plan to use the JavaScript display mode and display the sketch in your HTML 5 compatible browser.

Listing 1 also shows the declaration of three instance variables.

Listing 1. Beginning of the class named Pr0120a.

Figure

```
//@pjs preload required for JavaScript version in browser.
/* @pjs preload="Pr0120a.jpg"; */

PImage img;
PFont font;
```

```
Pr0120aRunner obj;
```

Listing 1. Beginning of the class named Pr0120a.

The setup method

The **setup** method is shown in its entirety in Listing 2 .
Listing 2. The setup method.
Figure

```
void setup(){
  size(400,344);
  frameRate(30);
  img = loadImage("Pr0120a.jpg");
  obj = new Pr0120aRunner();
  font = createFont("Arial",16,true);
}//end setup
```

Listing 2. The setup method.

Set the size of the display window

The **setup** method begins by calling the size method to set the size of the display window. Recall that the call to the **size** method must be the first line of code in the **setup** method. If the **size** method is not called, the display window will be given a default size of 100 x 100 pixels.

Recall also that if you plan to export your sketch in JavaScript mode or to export your sketch as a stand-alone application, you must use literal values as the parameters to the **size** method.

You should set the height of the display window to be at least 70 pixels greater than the height of the image to allow room to display the width, height, and pixel information

below the image as shown in Image 4 . Otherwise, the black text will be printed on the image and may not be visible, depending on the color of the image.

If you don't know the size of your image, simply make a guess. You will either get an output that displays the size of the image, as in Image 2 and Image 3 , or you will get an output that shows the image as in Image 1 and Image 4 . In either case, the width and height of the image will be displayed and you can use this information to update the call to the **size** method and recompile the sketch.

Set the font

The only other thing that is new in Listing 2 is the call to the createFont method.

This statement is required to prepare the sketch to draw the text characters on the display window as shown in Image 1 . I will leave it as an exercise for the student to study the documentation for the **createFont** method in whatever depth is necessary to understand it.

The draw method

The **draw** method is shown in its entirety in Listing 3 . As you can see, all the duties of the **draw** method were delegated to the **run** method belonging to an object of the class named **Pr0120aRunner** .

Listing 3. The draw method.

Figure

```
void draw(){
  obj.run();
}//end draw
```

Listing 3. The draw method.

You might be wondering why I didn't simply write that code in the **draw** method. The reason is that I frequently provide my students with input and output images *(such as*

Image 1 *and* *Image 4* *)* along with the driver code in Listing 10 . It is then the students responsibility to design and write the code in Listing 11 .

In other words, I provide a driver class along with the input and output images. It is the student's responsibility to design and write a class that will run in conjunction with the driver class to transform the input image into the output image.

The class named Pr0120aRunner

Beginning of the class

The class named **Pr0120aRunner** begins in Listing 4 .

Listing 4 also shows the beginning of the method named **run** that is called by the **draw** method in Listing 3 .

Listing 4. Beginning of the class named Pr0120aRunner.

Figure

```
class Pr0120aRunner{

  void run(){
    background(255);//white

    textFont(font,16);//Set the font size, and color
    fill(0);//black text

    loadPixels();//required
    img.loadPixels();//required

    float reD,greeN,bluE;//store color values here
    int ctr = 0;//output pixel array counter
```

Listing 4. Beginning of the class named Pr0120aRunner.

The **run** method begins by setting the background of the display window to white, and then takes care of some more housekeeping details regarding fonts.

Daniel Shiffman tells us about the requirement to use the loadPixels method and the updatePixels method in his tutorial titled Images and Pixels . I will leave it as an exercise for the student to study and understand that material.

Finally, Listing 4 declares some local variables that will be used later.

Check for size problems

The code in Listing 5 checks to confirm that the image will fit in the display window. If not, one of the messages shown in Image 2 and Image 3 is displayed in place of the image.

Listing 5. Check for size problems.

Figure

```
//Display error message in place of image if the
// image won't fit in the display window.
if(img.width > width){
  text("--Image too wide--",10,20);
  text("Image width: " + img.width,10,40);
  text("Display width: " + width,10,60);
}else if(img.height > height){
  text("--Image too tall--",10,20);
  text("Image height: " + img.height,10,40);
  text("Display height: " + height,10,60);
}else{
  //Copy pixel colors from the input image to the
  // display image.
```

Listing 5. Check for size problems.

Copy from input to output

If the image will fit in the display window, the code in Listing 6 is executed to begin the process of copying the pixel colors from the input image to the display image.

Listing 6. Copy from input to output.

Figure

```
for(int cnt = 0;cnt < img.pixels.length;cnt++){
  //Get and save RGB color values for current pixel.
  reD = red(img.pixels[cnt]);
  greeN = green(img.pixels[cnt]);
  bluE = blue(img.pixels[cnt]);

  //Normally some sort of image processing algorithm
  // would be implemented here.

  //Construct output pixel color
  color c = color(reD, greeN, bluE);
```

Listing 6. Copy from input to output.

An object of the PImage class

Although the syntax is unconventional, the call to the loadImage method in Listing 2 instantiates an object of the class named PImage and stores the object's reference in an instance variable named **img** . That object contains the pixel data extracted from the image file.

An object of the **PImage** class provides several methods that can be used to manipulate the image. In addition, an object of the **PImage** class contains fields for the **width** and **height** of the image, as well as an array named **pixels[]** that contains the values for every pixel in the image.

The arrays named pixels

The array containing the pixel data for the input image in this sketch can be accessed as **img.pixels[]** .

There is a similar pixel array that contains the pixel data for the output display window. That array can be accessed simply as **pixels[]** .

The data stored in the elements of these arrays is of the Processing primitive type color .

One-dimensional arrays

Although the pixels in an image can be thought of as a residing on a two-dimensional grid, the pixels are stored in these one-dimensional arrays row-by-row. In other words, the array element that is accessed as **pixels[0]** contains **color** data for the upper leftmost pixel in the image. The last element in the array contains **color** data for the bottom rightmost pixel in the image.

Extracting RGB color component values

The **Pr0120a** class inherits method named red , green , and blue .. These methods expect to receive a single parameter of type **color** , and return the value of the corresponding RGB component as type **float** . By default, the color component values range between 0.0 and 255.0.

The property named length and a for loop

Every array object in Java has a property named **length** whose value is equal to the number of elements in the array. This value is very useful in the conditional clause of loops that are used to traverse an entire array.

Listing 6 shows the beginning of a **for** loop that

- Extracts color component values from every pixel in the input image.
- Uses those values to construct color values and insert them into the elements of the array containing the pixel data for the output display.

This process is complicated somewhat by the fact that the actual output image may be wider than the input image. This is indicated by the white space on the right side of Image 1 . In other words, it is necessary to take this into account when computing the index of the output array element that is to receive the color from an input array element.

Store colors in the output array

The code that accomplishes that is shown in Listing 7 .
Listing 7. Store colors in the output array.

Figure

```
      if(width >= img.width){
        if((cnt % img.width == 0) && (cnt != 0)){
          //Compensate for excess display width by
          // increasing the output counter.
          ctr += (width - img.width);
        }//end if
        //Store the pixel in the output pixel array
        // and increment the output counter.
        pixels[ctr] = c;
        ctr++;
      }//end if
    }//end for loop

    updatePixels();//required

  }//end else
```

Listing 7. Store colors in the output array.

Although this isn't rocket science, you may need to get a pencil and paper and draw some diagrams in order to understand exactly how the code in Listing 7 work. I will leave that as an exercise for the student.

Listing 7 also signals the end of the **else** clause that began in Listing 5 .

Display pixel information

The code in Listing 8 tests to determine if a mouse button is pressed, and if so, calls the method named **displayPixelInfo** to cause the information shown at the bottom of Image 1 to be displayed.

If you remove the **if** statement and simply call the **displayPixelInfo** method at this point without regard to the mouse buttons, this information will be displayed any time that the mouse pointer is inside the display window.

Listing 8. Display pixel information.

Figure

```
  //Display pixel information only if a mouse button
```

```
  // is pressed.
  if(mousePressed){
    displayPixelInfo(img);
  }//end if
}//end run
```

Listing 8. Display pixel information.

Listing 8 also signals the end of the **run** method.

The method named displayPixelInfo

The method named **displayPixelInfo** is shown in its entirety in Listing 9 .

This method displays coordinate and color information for the pixel at the current mouse pointer location. It also displays the width and the height of the image.
Listing 9. The method named displayPixelInfo.

Figure

```
void displayPixelInfo(PImage image){
  //Protect against mouse being outside the frame
  if((mouseX < width) && (mouseY < height) &&
     (mouseX >= 0) && (mouseY >= 0)){

    //Get and display the width and height of the
    // image.
    text("Width: " + image.width + "  Height: " +
                     image.height,10,height - 50);

    //Get and display coordinates of mouse pointer.
    text("X: " + mouseX + ",  Y: " + mouseY,10,
                                      height - 30);

    //Get and display color data for the pixel at the
    // mouse pointer.
    text("R: " + red(pixels[mouseY*width+mouseX]) +
         "  G: " + green(pixels[mouseY*width+mouseX]) +
         "  B: " + blue(pixels[mouseY*width+mouseX]),
```

```
                                        10,height - 10);
    }//end if
  }//end displayPixelInfo
}//end class Pr0120aRunner
```

Listing 9. The method named displayPixelInfo.

Calls to the text method

The new material in Listing 9 is the set of repeated calls to the text method. There are many overloaded versions of this method. In a nutshell, the version used in Listing 9 displays the String given by the first parameter at a location in the display window specified by the second and third parameters.

The individual lines of text, as shown in Image 1 , are positioned relative to the bottom of the display window. Therefore, if there is insufficient blank space at the bottom of the window *(70 vertical pixels)*, the text will be drawn on the bottom of the image.

Converting from two dimensions to one dimension

You may also need some guidance relative to the use of the following expression in Listing 9 :

mouseY * width + mouseX

This is the expression that is used to extract color data from the one-dimensional pixel array for a pixel at a given X,Y coordinate position.

Listing 9 also signals the end of the class named **Pr0120aRunner** .

Run the program

I encourage you to copy the code from Listing 10 and Listing 11 and paste it into your PDE. Be sure to put the code from Listing 10 in the leftmost tab.

Don't forget to put an image file of your choice in a folder named **data** that is a child of the folder that contains the files with the .pde extension. You will need to edit the code

from Listing 10 to change the name of the image file in *two different places* . Change the name from **Pr0120a.jpg** to the name of your file.

Run the sketch and observe the results. Experiment with the code. Make changes, run the sketch again, and observe the results of your changes. Make certain that you can explain why your changes behave as they do.

Don't forget to also create and run the JavaScript version of your sketch in your HTML 5 compatible browser.

Click here to view the JavaScript version of the sketch discussed in this module in your HTML 5 compatible browser.

If you have a programmable Android device , try creating and running the Android version of your sketch in your Android device.

Also try creating and running the stand-alone version of the sketch by selecting **Export Application** from the **File** menu while in **Java** mode.

Summary

In this module, you learned how to write an image explorer sketch that can be used determine the coordinates and RGB color values of any pixel in an image by pointing to the pixel with a mouse. The sketch can also be used to determine the dimensions of the image.

Click here to view the JavaScript version of the sketch discussed in this module in your HTML 5 compatible browser.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Pr0120-Image Explorer
- File: Pr0120.htm
- Published: 02/25/13

Note: Disclaimers:
Financial : Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation :: I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listing

Complete listings of the code discussed in this module are provided in Listing 10 and Listing 11 .
Listing 10. Pr0120a.pde.

Figure

```
/*Pr0120a.pde
Copyright 2013, R.G.Baldwin
Program illustrates how to write an image explorer sketch
that will display the coordinates of the mouse pointer
along with the RGB color values of the pixel at the mouse
pointer.

Also displays the width and height of the image.

Displays an error message in place of the image if the
image is wider or taller than the output display window.
**********************************************************/
//@pjs preload required for JavaScript version in browser.
/* @pjs preload="Pr0120a.jpg"; */

PImage img;
PFont font;
```

```
Pr0120aRunner obj;
void setup(){
  //Make the height of the frame at least 70 pixels
  // greater than the height of the image to allow room
  // to display width, height, and pixel information.
  // Otherwise, the black text may not be visible,
  // depending on the image colors. If the height of the
  // frame is less than the height of the image, an error
  // message is displayed in place of the image.
  //Make width of the frame at least as wide as the
  // width of the image. Otherwise an error message will
  // be displayed in place of the image.
  size(400,344);
  frameRate(30);
  img = loadImage("Pr0120a.jpg");
  obj = new Pr0120aRunner();
  font = createFont("Arial",16,true);
}//end setup
//-------------------------------------------------------//
void draw(){
  obj.run();
}//end draw
```

Listing 10. Pr0120a.pde.

Listing 11. Class Pr0120aRunner.

Figure

```
class Pr0120aRunner{

  void run(){
    background(255);//white

    textFont(font,16);//Set the font size, and color
    fill(0);//black text

    loadPixels();//required
    img.loadPixels();//required
```

```
float reD,greeN,bluE;//store color values here
int ctr = 0;//output pixel array counter

//Display error message in place of image if the
// image won't fit in the display window.
if(img.width > width){
  text("--Image too wide--",10,20);
  text("Image width: " + img.width,10,40);
  text("Display width: " + width,10,60);
}else if(img.height > height){
  text("--Image too tall--",10,20);
  text("Image height: " + img.height,10,40);
  text("Display height: " + height,10,60);
}else{
  //Copy pixel colors from the input image to the
  // display image.
  for(int cnt = 0;cnt < img.pixels.length;cnt++){
    //Get and save RGB color values for current pixel.
    reD = red(img.pixels[cnt]);
    greeN = green(img.pixels[cnt]);
    bluE = blue(img.pixels[cnt]);

    //Normally some sort of image processing algorithm
    // would be implemented here.

    //Construct output pixel color
    color c = color(reD, greeN, bluE);

    if(width >= img.width){
      if((cnt % img.width == 0) && (cnt != 0)){
        //Compensate for excess display width by
        // increasing the output counter.
        ctr += (width - img.width);
      }//end if
      //Store the pixel in the output pixel array
      // and increment the output counter.
      pixels[ctr] = c;
      ctr++;
    }//end if
  }//end for loop
  updatePixels();//required
}//end else
```

```
    //Display pixel information only if a mouse button
    // is pressed.
    if(mousePressed){
      displayPixelInfo(img);
    }//end if
  }//end run

  //-------------------------------------------------
-//
  //Method to display coordinate and pixel color info at
  // the current mouse pointer location. Also displays
  // width and height information about the image.
  void displayPixelInfo(PImage image){
    //Protect against mouse being outside the frame
    if((mouseX < width) && (mouseY < height) &&
       (mouseX >= 0) && (mouseY >= 0)){

      //Get and display the width and height of the
      // image.
      text("Width: " + image.width + "  Height: " +
                       image.height,10,height - 50);

      //Get and display coordinates of mouse pointer.
      text("X: " + mouseX + ",  Y: " + mouseY,10,
                                       height - 30);

      //Get and display color data for the pixel at the
      // mouse pointer.
      text("R: " + red(pixels[mouseY*width+mouseX]) +
           "  G: " + green(pixels[mouseY*width+mouseX]) +
           "  B: " + blue(pixels[mouseY*width+mouseX]),
                                    10,height - 10);
    }//end if
  }//end displayPixelInfo
}//end class Pr0120aRunner
```

Listing 11. Class Pr0120aRunner.

-end-

Pr0130-Introduction to Image Processing Algorithms
The purpose of this module is to introduce you to pixel-based image processing algorithms, similar to those that you might find in commercial image editing software such as Photoshop.

Table of Contents

Preface

This module is one in a collection of modules designed to teach you about the Processing open source programming environment.

Processing is a Java-based OOP programming environment for people who want to create images, animations, and interactions.

The purpose of this module is to introduce you to pixel-based image processing algorithms, similar to those that you might find in commercial image editing software such as Photoshop .

Click here to view the JavaScript version of the sketch discussed in this module in your HTML 5 compatible browser.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

Images

- Image 1 . The raw image displayed in an image explorer.
- Image 2 . The modified image displayed in an image explorer.

Listings

- Listing 1 . Copy pixel colors from the input image to the display image.
- Listing 2 . The remainder of the for loop.
- Listing 3 . The remainder of the run method.
- Listing 4 . Class Pr0130a.
- Listing 5 . Class Pr0130aRunner.

Preview

The program that I will explain in this module is designed to be used as a preliminary test of the student's understanding of programming using Java and Processing Development Environment *(PDE)* .

If this were a real test, the student would be provided an image file named **Pr0130a.jpg** along with a pair of stand-alone Processing applications. Those applications would produce the raw image and a modified version of the raw image, each in an image explorer, as shown in Image 1 and Image 2 . *(Click here to view the modified image version of the sketch in your HTML 5 compatible browser.)*

Deduce the algorithm

The first requirement would be for the student to examine the raw image shown in the image explorer window in Image 1 and to deduce the algorithm required to transform that image into the modified image shown in the image explorer window in Image 2 .

Implement the algorithm

The second requirement would be for the student to implement the algorithm once it is established. Among other things, this would require that the student be able to:

- Load an image from an image file.
- Modify the pixels in the image according to the algorithm.
- Write one sketch to display the raw image in an image explorer window before it is modified.
- Write another sketch to display the modified image in another image explorer window after it is modified.

(Instructions for writing an image explorer sketch were provided in Pr0120-Image Explorer .)

Program output

Image 1 shows the raw image being displayed in an image explorer window. Note the red, green, and blue color component values and the coordinates of the pixel to which those colors belong.

Image 1. The raw image displayed in an image explorer.

Figure

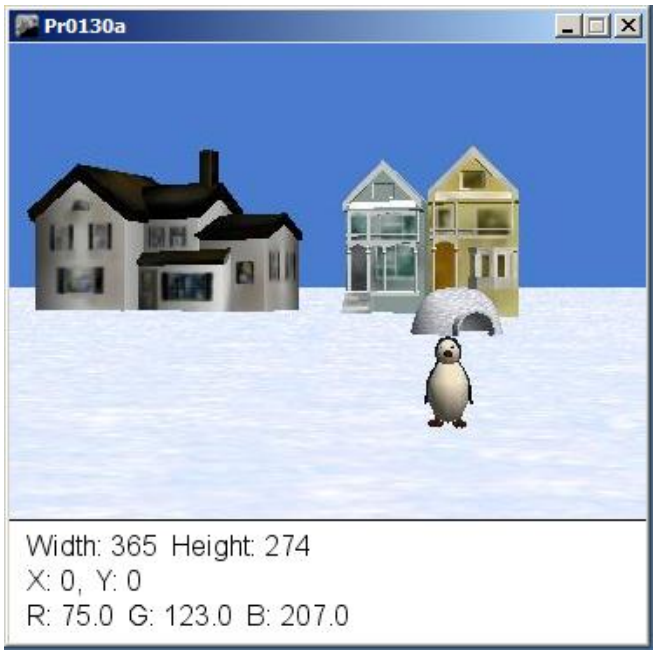


Image 1. The raw image displayed in an image explorer.

Image 2 shows the modified image in an image explorer window. Once again, note the red, green, and blue color component values and the coordinates of the pixel to which those colors belong.

Image 2. The modified image displayed in an image explorer.

Figure

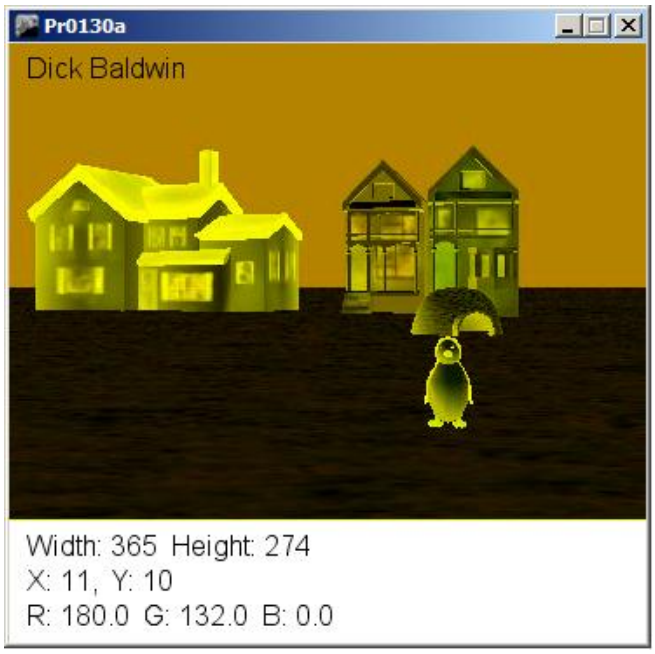


Image 2. The modified image displayed in an image explorer.

The algorithm

The algorithm required to transform the image from Image 1 to Image 2 is:

- Set the blue color value for every pixel to zero.
- Invert the red and green color values for every pixel.

A color value is inverted by subtracting the color value from 255.

Obvious that the blue color value is reduced to zero

By exploring the pixel colors at several different locations, it should be obvious to the student that the blue pixel value has been set to zero for every pixel in the modified image.

Color inversion is not quite so obvious

Deducing that the red and green colors in the output pixels are the inverse of the red and green colors in the input image isn't as straightforward. However, color inversion is one of the most common forms of image color manipulation, so a little research on the web should suffice to figure it out. I have also published several online tutorials that involve color inversion.

The implementation of the algorithm will be explained below.

Discussion and sample code

Will explain in fragments

This sketch consists of two classes, which are provided in Listing 4 and Listing 5 .

I will break the sketch down and explain it in fragments.

The driver class named Pr0130a

The driver class named **Pr0130a** , which is shown in its entirety in Listing 4 , is almost exactly like the class that I explained in Pr0120-Image Explorer . Therefore, I won't repeat that explanation in this module.

The runner class named Pr0130aRunner

The runner class named **Pr0130aRunner** is shown in Listing 5 . This class is only slightly different from the class that I explained in Pr0120-Image Explorer . Therefore, I will explain only those portions of the class that are different in this module.

The first difference

The first difference begins within the **for** loop shown in Listing 1 . Recall that in the original version in Pr0120-Image Explorer , the code in the **for** loop copied pixel colors from the input image to the output image taking the different widths of the images into account.

This version is essentially the same except that in this version, the code inverts the red and green color values and sets the blue color value to zero before inserting the pixel color into the output image.

Listing 1. Copy pixel colors from the input image to the display image.

Figure

```
//Copy pixel colors from the input image to the
// display image.
for(int cnt = 0;cnt < img.pixels.length;cnt++){
  //Get and save RGB color values for current pixel.
  reD = red(img.pixels[cnt]);
  greeN = green(img.pixels[cnt]);
  bluE = blue(img.pixels[cnt]);

  //color c = color(reD, greeN, bluE);//raw
  color c = color(255-reD, 255-greeN, 0);//modified
```

Listing 1. Copy pixel colors from the input image to the display image.

Construct output pixel color

You can selectively enable and disable the last two lines of code in Listing 1 to cause the output image to be either the raw image *(as in Image 1)* or a modified version of the raw image *(as in Image 2)*. As explained earlier , the red and green color components are inverted and the blue color component is set to zero in the last line of code in Listing 1 .

The remainder of the for loop

The code in Listing 2 is the same as in *Pr0120-Image Explorer* . It is included here to provide continuity in the discussion.

Listing 2. The remainder of the for loop.

Figure

```
if(width >= img.width){
  if((cnt % img.width == 0) && (cnt != 0)){
    //Compensate for excess display width by
    // increasing the output counter.
    ctr += (width - img.width);
```

```
          }//end if
          //Store the pixel in the output pixel array
          // and increment the output counter.
          pixels[ctr] = c;
          ctr++;
        }//end if
      }//end for loop
      updatePixels();//required
    }//end else
```

Listing 2. The remainder of the for loop.

The remainder of the run method

The code in Listing 3 is only slightly different from the code in *Pr0120-Image Explorer* .

Listing 3. The remainder of the run method.

Figure

```
    //Display the author's name on the output.
    text("Dick Baldwin",10,20);

    //Disable the requirement to press a mouse button to
    // display information about the pixel.
//    if(mousePressed){
      displayPixelInfo(img);
//    }//end if
  }//end run
```

Listing 3. The remainder of the run method.

The call to the **text** method was inserted in Listing 3 to display my name in the upper-left corner of Image 2 .

The call to the **displayPixelInfo** method was removed from the **if** statement in Listing 3 . This causes the pixel information shown at the bottom of Image 2 to be displayed any time that the mouse pointer is inside the display window with no requirement to press a mouse button to display the information.

(The information is actually displayed all of the time based on the last known location of the mouse pointer. At the beginning, the mouse pointer is assumed to be at coordinates 0,0. After that, if the mouse enters and then leaves the output display window, the last known location is the point on the edge where it left the window.)

The remainder of the class named class Pr0130aRunner

The remainder of the class named class **Pr0130aRunner** is essentially the same as before. I explained the entire class *in Pr0120-Image Explorer* and won't repeat that explanation in this module.

Run the program

I encourage you to copy the code from Listing 4 and Listing 5 into your PDE. Be sure to put the code from Listing 4 in the leftmost tab.

Don't forget to put an image file of your choice in a folder named **data** that is a child of the folder that contains the files with the .pde extension. You will need to edit the code from Listing 4 to change the name of the image file in *two different places* . Change the name from **Pr0130a.jpg** to the name of your file.

Run the sketch and observe the results. Experiment with the code. Make changes, run the sketch again, and observe the results of your changes. Make certain that you can explain why your changes behave as they do.

Don't forget to also create and run the JavaScript version of your sketch in your HTML 5 compatible browser.

Click here to view my JavaScript version of the sketch in your HTML 5 compatible browser.

If you have a programmable Android device , try creating and running the Android version of your sketch in your Android device.

Also try creating and running the stand-alone version of the sketch by selecting **Export Application** from the **File** menu while in **Java** mode.

Summary

This module introduced you to pixel-based image processing algorithms, similar to those that you might find in commercial image editing software such as Photoshop .

Click here to view the JavaScript version of the sketch discussed in this module in your HTML 5 compatible browser.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Pr0130-Introduction to Image Processing Algorithms
- File: Pr0130.htm
- Published: 02/26/13

Complete program listing

Complete listings of the classes discussed in this module are provided in Listing 4 and Listing 5 .

Listing 4. Class Pr0130a.

Figure

```
/*Pr0130a.pde
Copyright 2013, R.G.Baldwin

Program illustrates how to write a relatively simple image
processing algorithm and how to display the output in
an image explorer. The image explorer displays the
coordinates of the mouse pointer along with the RGB color
values of the pixel at the mouse pointer. Also displays
the width and height of the image.

Displays an error message in place of the image if the
image is wider or taller than the output display window.
**********************************************************/
//@pjs preload required for JavaScript version in browser.
/* @pjs preload="Pr0130a.jpg"; */

PImage img;
PFont font;

Pr0130aRunner obj;
void setup(){
  //This size matches the width of the image and allows
  // space below the image to display the text information.
  size(365,344);
  frameRate(30);
  img = loadImage("Pr0130a.jpg");
  obj = new Pr0130aRunner();
  font = createFont("Arial",16,true);
}//end setup
//--------------------------------------------------------//
void draw(){
  obj.run();
}//end draw
```

Listing 4. Class Pr0130a.

Listing 5. Class Pr0130aRunner.

Figure

```
class Pr0130aRunner{

  void run(){
    background(255);//white

    textFont(font,16);//Set the font size, and color
    fill(0);//black text

    loadPixels();//required
    img.loadPixels();//required

    float reD,greeN,bluE;//store color values here
    int ctr = 0;//output pixel array counter

    //Display error message in place of image if the
    // image won't fit in the display window.
    if(img.width > width){
      text("--Image too wide--",10,20);
      text("Image width: " + img.width,10,40);
      text("Display width: " + width,10,60);
    }else if(img.height > height){
      text("--Image too tall--",10,20);
      text("Image height: " + img.height,10,40);
      text("Display height: " + height,10,60);
    }else{
      //Copy pixel colors from the input image to the
      // display image.
      for(int cnt = 0;cnt < img.pixels.length;cnt++){
        //Get and save RGB color values for current pixel.
        reD = red(img.pixels[cnt]);
        greeN = green(img.pixels[cnt]);
        bluE = blue(img.pixels[cnt]);

        //Construct output pixel color
        //Selectively enable and disable the following two
        // statements to display either the raw image, or
        // a modified version of the raw image where the
        // red and green color components have been
```

```
        // inverted and the blue color component has been
        // set to zero.
        //color c = color(reD, greeN, bluE);//raw
        color c = color(255-reD, 255-greeN, 0);//modified

        if(width >= img.width){
          if((cnt % img.width == 0) && (cnt != 0)){
            //Compensate for excess display width by
            // increasing the output counter.
            ctr += (width - img.width);
          }//end if
          //Store the pixel in the output pixel array
          // and increment the output counter.
          pixels[ctr] = c;
          ctr++;
        }//end if
      }//end for loop
      updatePixels();//required
    }//end else

    //Display the author's name on the output.
    text("Dick Baldwin",10,20);

    //Disable the requirement to press a mouse button to
    // display information about the pixel.
//    if(mousePressed){
      displayPixelInfo(img);
//    }//end if
  }//end run

  //-----------------------------------------------
-//
  //Method to display coordinate and pixel color info at
  // the current mouse pointer location. Also displays
  // width and height information about the image.
  void displayPixelInfo(PImage image){
    //Protect against mouse being outside the frame
    if((mouseX < width) && (mouseY < height) &&
       (mouseX >= 0) && (mouseY >= 0)){

      //Get and display the width and height of the
      // image.
      text("Width: " + image.width + "  Height: " +
```

```
                    image.height,10,height - 50);

      //Get and display coordinates of mouse pointer.
      text("X: " + mouseX + ",  Y: " + mouseY,10,
                                      height - 30);

      //Get and display color data for the pixel at the
      // mouse pointer.
      text("R: " + red(pixels[mouseY*width+mouseX]) +
           "  G: " + green(pixels[mouseY*width+mouseX]) +
           "  B: " + blue(pixels[mouseY*width+mouseX]),
                                    10,height - 10);
    }//end if
  }//end displayPixelInfo
}//end class Pr0130aRunner
```

Listing 5. Class Pr0130aRunner.

-end-

Pr0140-A space-wise linear pixel-modification algorithm
The purpose of this module is to teach you: (1) how to develop a template sketch for implementing pixel modification algorithms, and (2) how to implement a space-wise linear pixel modification algorithm.

Table of Contents

Preface

This module is one in a collection of modules designed to teach you about the Processing open source programming environment.

Processing is a Java-based OOP programming environment for people who want to create images, animations, and interactions.

The purpose of this module is to teach you:

1. How to develop a template sketch for implementing pixel modification algorithms, and
2. How to implement a space-wise linear pixel modification algorithm.

Click here to view the JavaScript version of the sketch discussed in this module in your HTML 5 compatible browser.

Viewing tip

I recommend that you open another copy of this module in a separate browser window and use the following links to easily find and view the images and listings while you are reading about them.

Images

- Image 1 . The raw image.
- Image 2 . The modified image.

Listings

- Listing 1 . Beginning of the class and the run method.
- Listing 2 . Beginning of the processPixels method.
- Listing 3 . Beginning of a for loop.
- Listing 4 . Compute the scale factor for the column.
- Listing 5 . Compute a new color.
- Listing 6 . Store modified pixel color in the output pixel array.
- Listing 7 . Pr0140a.pde.
- Listing 8 . Pr0140aRunner.pde.

Preview

The algorithm

The pixel modification algorithm that I will implement in this module can be described as follows:

Scale the blue and green color components of each pixel by a scale factor that is less than or equal to 1.0.

The green scale factor:

- Is equal to 1.0 on the left side of the image
- Is equal to 0.0 on the right side of the image
- Decreases linearly with distance going from left to right across the image.

The blue scale factor

- Is 0.0 on the left side of the image
- Is 1.0 on the right side of the image

- Increases linearly with distance going from left to right across the image.

The value of the red color component is not modified.

The output images

Depending on whether or not pixel modification is enabled before compiling the program, the sketch will produce one of the two images shown in Image 1 and Image 2 .

The raw image is shown in Image 1 .
Image 1. The raw image.
Figure

Image 1. The raw image.

The modified image is shown in Image 2 .
Image 2. The modified image.
Figure

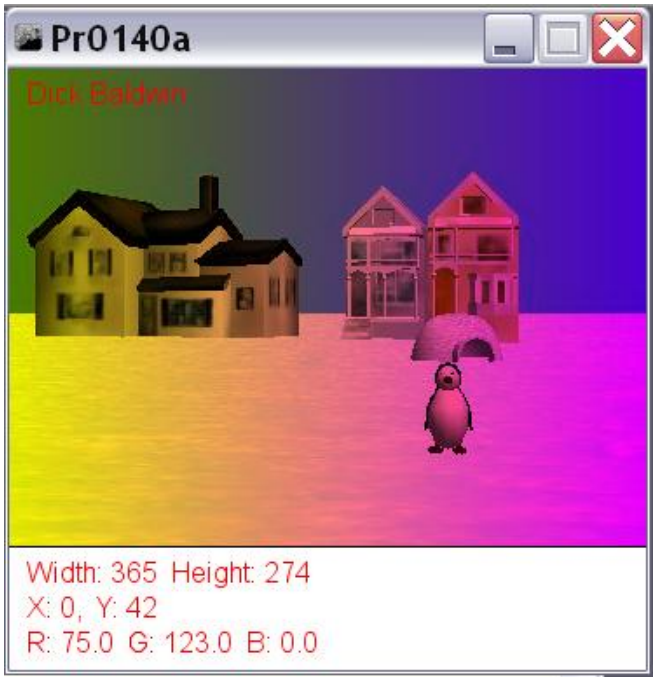


Image 2. The modified image.

Discussion and sample code

This sketch consists of two classes:

- Pr0140a *(see Listing 7)*
- Pr0140aRunner *(see Listing 8)*

The code in Listing 7 is the code that needs to be in the leftmost tab of the PDE.

Will explain in fragments

I will break this code down and explain it in fragments. As I go along, I will be explaining how this code forms a template for the development of other pixel modification algorithms in future modules.

The driver class named Pr0140a

The driver class that appears in the leftmost tab in the PDE hasn't changed significantly since I first explained it in Pr0120-Image Explorer . Therefore, I won't repeat that explanation here.

The runner class named Pr0140aRunner

Aside from the specific implementation of the pixel modification algorithm, the major difference between this sketch and the sketches since Pr0120-Image Explorer is that I have made the code more modular.

For example, I have attempted to isolate those portions of the template that are likely to change from one algorithm go the next from those portions that are less likely to change. That will make it easier to explain how the code for one algorithm differs from the code for other algorithms.

The beginning of the class and the run method

Listing 1 shows the beginning of the class and the modified **run** method. It is unlikely that this code will change much from one algorithm to the next.

Listing 1. Beginning of the class and the run method.

Figure

```
class Pr0140aRunner{
  //The following instance variable is used to set the
  // color of the appropriate pixel in the output display
  // window.
  int ctr = 0;//output pixel array counter

  void run(){
    background(255);//white
    textFont(font,16);//Set the font size
    fill(255,0,0);//Set the text color to red

    //Display error message in place of image if the
    // image won't fit in the display window.
    if(img.width > width){
      text("--Image too wide--",10,20);
      text("Image width: " + img.width,10,40);
      text("Display width: " + width,10,60);
```

```
    }else if(img.height > height){
      text("--Image too tall--",10,20);
      text("Image height: " + img.height,10,40);
      text("Display height: " + height,10,60);
    }else{
      //The image will fit in the output window.

      //Call a method that will apply a specific
      // pixel-modification algorithm and write the
      // modified pixel colors into the output window.
      processPixels();
    }//end else

    //Display the author's name on the output in the font
    // size and text color defined above.
    text("Dick Baldwin",10,20);

    //Display information about the pixel being pointed
    // to with the mouse. Display near the bottom of the
    // output window.
    displayPixelInfo(img);
  }//end run
```

Listing 1. Beginning of the class and the run method.

Major differences in the run method

The first major difference between this and previous versions of the **run** method is the delegation of pixel manipulation operations to the method named **processPixels** . Moving those operations to a separate method will decrease the likelihood that it will be necessary to modify the code in Listing 1 to implement different algorithms.

I will identify the second major change later.

Beginning of the processPixels method

The **processPixels** method, which begins in Listing 2 , applies a pixel modification algorithm that causes the green and blue color values to be scaled on a linear basis moving from left to right across the image.

Listing 2. Beginning of the processPixels method.

Figure

```
void processPixels(){
  loadPixels();//required
  img.loadPixels();//required
  float reD,greeN,bluE;//store color values here
  ctr = 0;//initialize output pixel array counter
```

Listing 2. Beginning of the processPixels method.

The code in Listing 2 :

- Does some preliminary housekeeping regarding pixels, *(which you have seen before)* .
- Declares local variables for storage of red, green, and blue color values.
- Initializes the counter that is used to position color values in the output pixel array.

Beginning of a for loop

Listing 3 shows the beginning of a **for** loop that is used to process every pixel in the input pixel array.

Listing 3. Beginning of a for loop.

Figure

```
//Process each pixel in the input image.
for(int cnt = 0;cnt < img.pixels.length;cnt++){
  //Get and save RGB color values for current pixel.
  reD = red(img.pixels[cnt]);
  greeN = green(img.pixels[cnt]);
  bluE = blue(img.pixels[cnt]);
```

Listing 3. Beginning of a for loop.

Get and save color values

The code in Listing 3 gets and saves the RGB color values for the current pixel.

Compute the column number

This algorithm applies the same scale factor to every pixel in each column. Therefore, it is necessary to identify the column to which a pixel belongs when the colors for the pixel are retrieved from the pixel array. However, it is not necessary to identify the row. *(We will get to that in later modules.)*

The code in Listing 4 uses the modulus operator to compute the column number based on the loop counter, *(which is the same as the input pixel array index)* .
Listing 4. Compute the scale factor for the column.

Figure

```
//Compute the column number and use it to compute
// the linear scale factors that will be applied to
// the green and blue color values.
int col = cnt%img.width;
float greenScale = (float)(width - col)/width;
float blueScale = (float)(col)/width;
```

Listing 4. Compute the scale factor for the column.

Compute the scale factor for the column

The code in Listing 4 uses the column number to compute the two required scale factors.

If case you haven't recognized it, the expression for **greenScale** is the equation for a straight line that goes through 1.0 on the left side of the image and goes through 0.0 on the right side of the image.

Similarly, the expression for **blueScale** is the equation for a straight line that goes through 0.0 on the left and 1.0 on the right.

Compute a new color

Listing 5 computes a new color based on scaled versions of the green and blue input color values. The red color value is not modified.

Listing 5. Compute a new color.

Figure

```
      color colr =
              color(reD, greenScale*greeN,
blueScale*bluE);
      //Enable the following statement to override the
      // color modification and display the raw image in
      // the output window. Disable it to display the
      // modified image.
      //colr = color(reD, greeN, bluE);
```

Listing 5. Compute a new color.

If you enable the last statement in Listing 5 , the new output color will be identical to the old input color. This is useful when you need to produce an output image showing the unmodified input image as in Image 1 .

Store modified pixel color in the output pixel array

The second major difference between this and previous versions of the **run** method is the delegation of the code that stores the modified pixel color in the output pixel array to a separate method named **setOutputPixelColor** .

This was done because it is unlikely that the code needed to perform this operation will need to change from one algorithm to the next. As a result, most of the code that is likely to change from one algorithm to the next is confined to the method named **processPixels** .

The method named **setOutputPixelColor** is called in Listing 6 , which also signals the end of the **processPixels** method
Listing 6. Store modified pixel color in the output pixel array.
Figure

```
      setOutputPixelColor(cnt,colr);
    }//end for loop
    updatePixels();//required
  }//end processPixels
```

Listing 6. Store modified pixel color in the output pixel array.

The setOutputPixelColor method

The **setOutputPixelColor** method is shown in Listing 8 . The code in that method is the same as code that I have explained in earlier modules, so I won't repeat that explanation here.

The remainder of the Pr0140aRunner class

The remainder of the **Pr0140aRunner** class shown in Listing 8 is the same as code that I have explained in earlier modules. Therefore, there is nothing more to explain in this module.

Run the sketch

I encourage you to copy the code from Listing 7 and Listing 8 and paste it into your PDE. Be sure to put the code from Listing 7 in the leftmost tab.

Don't forget to put an image file of your choice in a folder named **data** that is a child of the folder that contains the files with the .pde extension. You will need to edit the code to change the name of the image file in *two different places* .

Run the sketch and observe the results. Experiment with the code. Make changes, run the sketch again, and observe the results of your changes. Make certain that you can

explain why your changes behave as they do.

Don't forget to also create and run the JavaScript version of your sketch in your HTML 5 compatible browser.

Click here to view the JavaScript version of the sketch discussed in this module in your HTML 5 compatible browser.

If you have a programmable Android device , try creating and running the Android version of your sketch in your Android device.

Also try creating and running the stand-alone version of the sketch by selecting **Export Application** from the **File** menu while in **Java** mode.

Summary

In this module, you learned:

1. How to develop a template sketch for implementing pixel modification algorithms, and
2. How to implement a space-wise linear pixel modification algorithm.

Click here to view the JavaScript version of the sketch discussed in this module in your HTML 5 compatible browser.

Miscellaneous

This section contains a variety of miscellaneous information.

Note: Housekeeping material

- Module name: Pr0140-A space-wise linear pixel-modification algorithm
- File: Pr0140.htm
- Published: 02/26/13

pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.
I also want you to know that, I receive no financial compensation from the Connexions website even if you purchase the PDF version of the module.
In the past, unknown individuals have copied my modules from cnx.org, converted them to Kindle books, and placed them for sale on Amazon.com showing me as the author. I neither receive compensation for those sales nor do I know who does receive compensation. If you purchase such a book, please be aware that it is a copy of a module that is freely available on cnx.org and that it was made and published without my prior knowledge.
Affiliation : I am a professor of Computer Information Technology at Austin Community College in Austin, TX.

Complete program listing

Complete listings of the classes discussed in this module are provided in Listing 7 and Listing 8 .
Listing 7. Pr0140a.pde.

Figure

```
/*Pr0140a.pde
Copyright 2013, R.G.Baldwin

This sketch can be used as a template for writing other
pixel modification algorithms.

The sketch illustrates a linear space wise pixel
modification algorithm in which the green and blue pixel
color values are scaled linearly as a function of the
distance of the pixel from the left side of the image.

The output is displayed in an image explorer.

The image explorer displays the coordinates of the mouse
pointer along with the RGB color values of the pixel at
the mouse pointer. It also displays the width and height
of the image.

The image explorer displays an error message in place of
the image if the image is wider or taller than the output
```

```
display window.
**************************************************************/
//@pjs preload required for JavaScript version in browser.
/* @pjs preload="Pr0140a.jpg"; */

PImage img;
PFont font;

Pr0140aRunner obj;
void setup(){
  //This size matches the width of the image and allows
  // space below the image to display the text information.
  size(365,344);
  frameRate(30);
  img = loadImage("Pr0140a.jpg");
  obj = new Pr0140aRunner();
  font = createFont("Arial",16,true);
}//end setup
//-------------------------------------------------------//
void draw(){
  obj.run();
}//end draw
```

Listing 7. Pr0140a.pde.

Listing 8. Pr0140aRunner.pde.

Figure

```
class Pr0140aRunner{
  //The following instance variable is used to set the
  // color of the appropriate pixel in the output display
  // window.
  int ctr = 0;//output pixel array counter

  void run(){
    background(255);//white
    textFont(font,16);//Set the font size
    fill(255,0,0);//Set the text color to red

    //Display error message in place of image if the
```

```
    // image won't fit in the display window.
    if(img.width > width){
      text("--Image too wide--",10,20);
      text("Image width: " + img.width,10,40);
      text("Display width: " + width,10,60);
    }else if(img.height > height){
      text("--Image too tall--",10,20);
      text("Image height: " + img.height,10,40);
      text("Display height: " + height,10,60);
    }else{
      //The image will fit in the output window.

      //Call a method that will apply a specific
      // pixel-modification algorithm and write the
      // modified pixel colors into the output window.
      processPixels();
    }//end else

    //Display the author's name on the output in the font
    // size and text color defined above.
    text("Dick Baldwin",10,20);

    //Display information about the pixel being pointed
    // to with the mouse. Display near the bottom of the
    // output window.
    displayPixelInfo(img);
  }//end run
  //-------------------------------------------------
-//

  //Apply a pixel modification algorithm that causes the
  // green and blue color values to be scaled on a linear
  // basis moving from left to right across the image.
  void processPixels(){
    loadPixels();//required
    img.loadPixels();//required
    float reD,greeN,bluE;//store color values here
    ctr = 0;//initialize output pixel array counter

    //Process each pixel in the input image.
    for(int cnt = 0;cnt < img.pixels.length;cnt++){
      //Get and save RGB color values for current pixel.
      reD = red(img.pixels[cnt]);
```

```
      greeN = green(img.pixels[cnt]);
      bluE = blue(img.pixels[cnt]);

      //Compute the column number and use it to compute
      // the linear scale factors that will be applied to
      // the green and blue color values.
      int col = cnt%img.width;
      float greenScale = (float)(width - col)/width;
      float blueScale = (float)(col)/width;

      //Compute a new color based on scaled versions of
      // the input color values. Don't modify the red
      // color value.
      color colr =
              color(reD, greenScale*greeN,
blueScale*bluE);
      //Enable the following statement to override the
      // color modification and display the raw image in
      // the output window. Disable it to display the
      // modified image.
      //colr = color(reD, greeN, bluE);

      //Store the modified pixel color in the output pixel
      // array.
      setOutputPixelColor(cnt,colr);
    }//end for loop
    updatePixels();//required
  }//end processPixels
  //-------------------------------------------------
-//

  //Method to set the color of a pixel in the output image
  // based on the input pixel counter, the output pixel
  // counter, the widths of the input and output images,
  // and the desired color. Deals with the possibility
that
  // the output display window is wider than the image
being
  // processed.
  void setOutputPixelColor(int cnt,color colr){
    if(width >= img.width){
      if((cnt % img.width == 0) && (cnt != 0)){
        //Compensate for excess display width by
```

```
        // increasing the output counter.
        ctr += (width - img.width);
      }//end if
      //Store the pixel in the output pixel array
      // and increment the output pixel counter.
      pixels[ctr] = colr;
      ctr++;
    }//end if
  }//end setOutputPixelColor

  //-------------------------------------------------
-//
  //Method to display coordinate and pixel color info at
  // the current mouse pointer location. Also displays
  // width and height information about the image.
  void displayPixelInfo(PImage image){
    //Protect against mouse being outside the frame
    if((mouseX < width) && (mouseY < height) &&
       (mouseX >= 0) && (mouseY >= 0)){

      //Get and display the width and height of the
      // image.
      text("Width: " + image.width + "  Height: " +
                       image.height,10,height - 50);

      //Get and display coordinates of mouse pointer.
      text("X: " + mouseX + ",  Y: " + mouseY,10,
                                       height - 30);

      //Get and display color data for the pixel at the
      // mouse pointer.
      text("R: " + red(pixels[mouseY*width+mouseX]) +
           "  G: " + green(pixels[mouseY*width+mouseX]) +
           "  B: " + blue(pixels[mouseY*width+mouseX]),
                                    10,height - 10);
    }//end if
  }//end displayPixelInfo
}//end class Pr0140aRunner
```

Listing 8. Pr0140aRunner.pde.

-end-